From Perception to Programs: Regularize, Overparameterize, and Amortize

Hao Tang [1] **Kevin Ellis** [1]

Abstract

We develop techniques for synthesizing neurosymbolic programs. Such programs mix discrete symbolic processing with continuous neural computation. We relax this mixed discrete/continuous problem and jointly learn all modules with gradient descent, and also incorporate amortized inference, overparameterization, and a differentiable strategy for penalizing lengthy programs. Collectedly this toolbox improves the stability of gradient-guided program search, and suggests ways of learning both how to parse continuous input into discrete abstractions, and how to process those abstractions via symbolic code.

1. Introduction

We seek steps toward AI systems that learn to symbolically process perceptual input. Consider, for example, a system which learns to infer the 3D structure of objects: starting from pixels, it must infer low-level symbols (curves, parts), and then organize them according to symbolic relationships (symmetry, part repetitions, part hierarchy). Or, consider a system which learns to control a moving object that navigates around obstacles: starting from sensory data (lidar, RGBD), it must first parse the world (into objects, proximities, freespace), and then compute trajectories using high-level computations (PID controllers, etc.). Similar perceptual-symbolic problems arise when learning structured world models from pixels, inferring instructions from natural language, or constructing visual analogies. We propose framing such tasks as **neurosymbolic program synthesis**: learning neural components that extract symbols from perception, and synthesizing programs to further process those symbols with more complex computations.

Our ultimate goal is to develop general methods that could, we hope, apply to challenging neurosymbolic tasks like those previously mentioned. We take the stance that symbols should be grounded in perception, and that symbol processing should be implemented by learnable program-like representations. However, we also propose that rather than hand-code a preordained set of primitive symbols, AI systems should learn to carve the perceptual world into their own discretization. What constitutes a 'symbol' may vary across domain and across datasets, and can be hard for human engineers to anticipate. By jointly learning the symbols, as well as synthesizing the programs that operate on them, we hope to side-step the pitfalls associated with hand-engineered representations.

Delivering on the above promises requires synthesizing neurosymbolic programs, which poses unique technical challenges. Unlike conventional programs, which are discrete, a neurosymbolic program has both continuous weights and discrete program structure, both of which must be synthesized. In addition to solving a mixed discrete/continuous problem, synthesizing a neurosymbolic program is severely underconstrained. It is under constrained because it is not clear what parts of the problem should be handled by symbolic processing, and what should be handled by neural networks. Because neural nets are universal function approximators, they can in theory satisfy any program learning problem, at least on the training data.

Our main technical contribution is a suite of methods for circumventing the above two challenges. We assume a multitask setup where the learner is exposed to a variety of neurosymbolic programming tasks. Having multiple tasks introduces extra constraints, and also allows learning across tasks how to search for programs. Hence multitasking can address both the ill-posed nature of the problem, and also the intractable search aspect due to the mixed discrete/continuous nature of the problem.

Concretely, our method trains a neural search policy to synthesize neurosymbolic programs. It uses a differentiable interpreter to backprop gradients from the desired program output all the way back to the parameters of the search policy. We overparametrize the program search space to ease continuous optimization, but this overparametrization leads to bloated programs with too much code, hence we regularize the length of the programs to produce concise, interpretable code. We therefore call our method ROAP (Regularize, Overparametrize, Amortize, for Programs).

[1]Cornell University. Correspondence to: Hao Tang <haotang@cs.cornell.edu>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

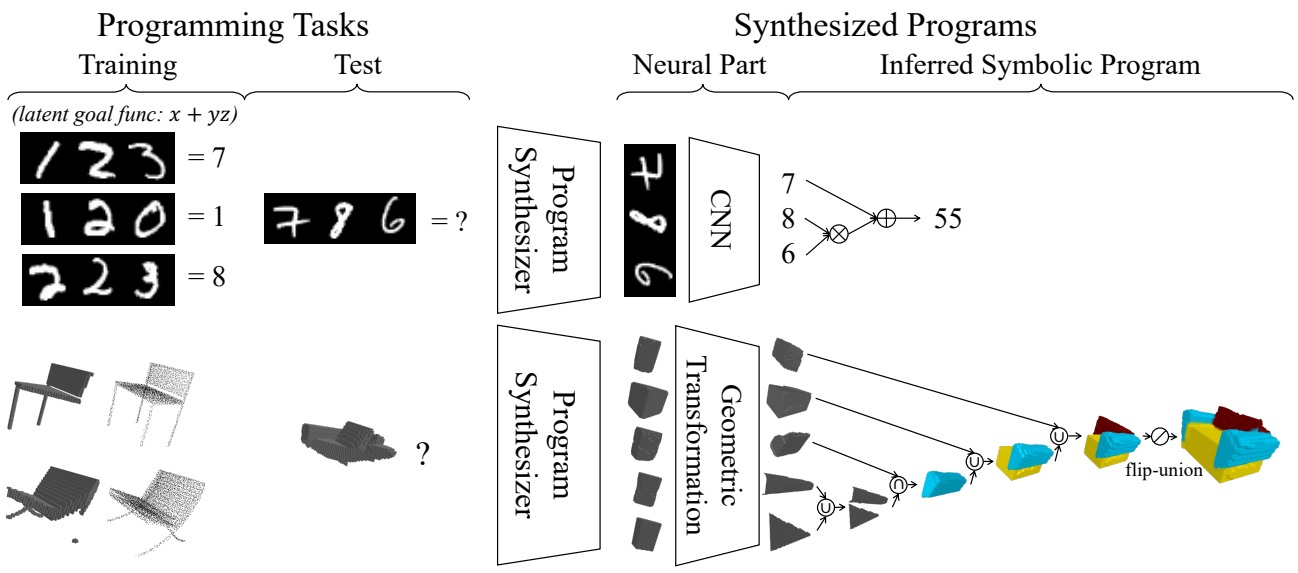


Figure 1. **Top, CIFAR-MATH domain.** System synthesizes symbolic equations, but instead of learning the equations from concrete numbers, it inputs CIFAR-10 images, where each image category (dog, cat, frog, ...) has been mapped to a digit from 0–9. Note: we show MNIST above for ease of understanding. **Bottom, graphics domain.** Given a partial observation of a 3D shape, system learns to infer a 3D graphics program completing the shape. Note: For CIFAR-MATH we show one task. For 3D we show several tasks.

We apply ROAP to two different domains (Fig. 1). Our CIFAR-MATH domain is a harder version of a classic proving ground for neural logic programming (Manhaeve et al., 2018), modified to include program synthesis. On it, we show that ROAP can synthesize arithmetic equations while at the same time learning to parse images into symbolic digits. Our 3D-Reconstruction domain involves synthesizing graphics programs that algebraically transform and combine neural geometric primitives, and can be used to decompose 3D shapes and infer missing geometry. In total our work makes the following contributions:

- A synthesis method for neurosymbolic programs. ROAP works without supervising on source code, and so doesn't require a training set of programs, unlike e.g. Codex (Chen et al., 2021a). This is important because it allows unsupervised learning over large datasets, even if those datasets are not designed for program synthesis. ROAP also does not require pretraining the 'neural' part of a neurosymbolic program, unlike (Chen et al., 2021b). ROAP *does* require a Domain Specific Language, which restricts the space of programs and imparts human prior knowledge.
- Comparison against four prior neurosymbolic synthesis methods (Shah et al., 2020; Gaunt et al., 2016; Valkov et al., 2018; Cui & Zhu, 2021). To the best of our knowledge, the neurosymbolic program synthesis field lacks direct comparisons among these prototypical methods.

2. Problem statement & Technical background

Definitions: Architecture, parameters, denotation. A neurosymbolic program has both a **symbolic program architecture** α, and also **continuous parameters** θ. Each architecture comes from a set $\mathcal{A}$ of possible architectures. We can instantiate a fixed architecture with different continuous parameters, and we write α_θ for the program with architecture α and parameters θ. We assume a **denotation operator** $[\![\cdot]\!]$, which takes a program α_θ and outputs what the program executes to. Generally, $[\![\alpha_\theta]\!]$ is a function.

An example of synthesizing a neurosymbolic program is optimizing for the architecture $\alpha \in \mathcal{A}$ and parameters $\theta \in \mathbb{R}^d$ minimizing a loss function over training data $\mathcal{D}$:

$$\alpha, \theta = \underset{\substack{\theta \in \mathbb{R}^d \\ \alpha \in \mathcal{A}}}{\arg\min} \sum_{(x,y)\in\mathcal{D}} \text{Loss}\big(y, [\![\alpha_\theta]\!](x)\big) \qquad (1)$$

This is challenging because it involves optimizing over discrete α (from combinatorially large $\mathcal{A}$) and continuous θ (which is potentially high dimensional). The trick of **relaxation** is to convert this mixed discrete-continuous problem into a purely continuous one, and then optimize with continuous methods. Intuitively, relaxations index the space of architectures using continuous weights that interpolate between discrete structures:

Definition: Relaxation. Architectures $\mathcal{A}$ and denotation $[\![\cdot]\!]$ admit a **k-dimensional relaxation** when the architectures are represented as k-dimensional vectors ($\mathcal{A} \subset \mathbb{R}^k$) and we

can take the denotation of any such k-dimensional vector, even ones not in $\mathcal{A}$, which means $\mathbb{R}^k \subseteq \text{domain}(\llbracket\cdot\rrbracket)$.

There are many relaxation approaches differing on what exactly the denotation means as an embedding 'interpolates' between architectures. Some relaxations define an approximate probabilistic semantics and interpret the k-dimensional vector as a vector of probabilities (Si et al., 2019; Gaunt et al., 2016; Chaudhuri & Solar-Lezama, 2010). Others use schemes reminiscent of fuzzy logic (Evans & Grefenstette, 2018), or form linear combinations of discrete subprograms (Sahoo et al., 2018). Either way, solving the relaxation typically proceeds by finding a continuous vector using gradient-based optimization, and then discretizing that vector to the closest symbolic architecture.

Amortized inference: Learning to search. The idea behind amortized inference (Gershman & Goodman, 2014) is to learn to search for programs ("infer" programs). Instead of directly optimizing over the space of programs, amortized inference in this context means optimizing a policy that probabilistically generates programs, conditioned on a particular programming task to solve. Typically the policy is trained across many tasks so that it learns to generate programs that solve each task (Devlin et al., 2017).

3. Method

We assume a training corpus of neurosymbolic programming tasks, $\mathcal{T}$. Each such task $t \in \mathcal{T}$ is specified by a dataset $\mathcal{D}_t$ of input-output pairs, (x, y). For example, in the CIFAR-MATH domain, each input x is a triple of CIFAR-10 images, while each output y is a real number, and the task t is a collection of (image-triple, scalar) input-outputs. In the 3D reconstruction domain, x is a point in 3D space and y is either 1 or 0, depending on if x is inside or outside the object, respectively; meanwhile t is voxels, possibly with missing or noisy data.

Amortization & Parameter Sharing. We start with an objective function for amortized inference that optimizes the parameters of a policy, ϕ, to increase the probability of generating a program architecture that has low loss. We also optimize the parameters θ of the neural networks invoked by these symbolic program architectures, ultimately minimizing $\mathcal{L}(\theta, \phi)$ shown below:

$$\mathcal{L}(\theta,\phi) = \mathop{\mathbb{E}}_{\substack{t\sim\mathcal{T}\\ \alpha\sim\pi_\phi(\cdot|t)}} \left[\sum_{(x,y)\in\mathcal{D}_t} \text{Loss}(y, \llbracket\alpha_\theta\rrbracket(x)) \right] \quad (2)$$

Already, this framing helps address one issue with synthesizing neurosymbolic programs: Each program can invoke learned neural networks, but only ones using *shared* parameters θ. Thus having multiple tasks introduces extra constraints on θ, preventing the system from solving everything with monolithic neural networks. The alternative possibility of optimizing task-specific continuous parameters $(\theta_t, \forall\, t \in \mathcal{T})$ would not introduce these extra constraints.

Gradient estimation via Relaxation. Learning to search for programs requires optimizing the search policy parameters ϕ. We implement our policies as neural networks, so we are interested in taking gradients of $\mathcal{L}$ with respect to ϕ (and also θ). While one could use a reinforcement-learning style approach, getting useful training signal from such methods would require serendipitously finding good program architectures from a randomly initialized policy. Absent pretraining, symbolic programs are hard to randomly guess correctly. How then can we get training off the ground?

At a high level, ROAP relaxes the symbolic program space; assumes that sampling a program architecture is equivalent to sampling an array of one-hot vectors from categorical distributions; and then finally uses Gumbel-Softmax to backprop through these categorical draws. In low-level detail, we assume the relaxed program semantics allow backpropagating through the denotation operator. However, we still have to pass gradients backward through the random sampling from the policy (expectation over $\alpha \sim \pi_\phi(\cdot \mid t)$ in Eq. 2). To do this, we assume each symbolic architecture α is encoded as C one-hot vectors,[1] notated $\{\alpha_c\}_{c=1}^{C}$, and the policy π_ϕ samples an architecture α by drawing from C categorical distributions with parameters $\{p_\phi^c(t)\}_{c=1}^{C}$:

$$\pi_\phi(\alpha \mid t) = \prod_{1\le c\le C} \text{Cat}\left(\alpha_c;\ p_\phi^c(t)\right) \quad (3)$$

This licenses rewriting the objective in Eq. 2 as

$$\mathop{\mathbb{E}}_{t\sim\mathcal{T}} \mathop{\mathbb{E}}_{\substack{\alpha_c\sim\text{Cat}(\cdot;\ p_\phi^c(t))\\ \forall\, 1\le c\le C}} \left[\sum_{(x,y)\in\mathcal{D}_t} \text{Loss}(y, \llbracket\alpha_\theta\rrbracket(x)) \right] \quad (4)$$

At this point we can deploy the well-known Gumbel-Softmax trick (Jang et al., 2016), which offers a low-variance approximation to the above expectation. Gumbel-Softmax perturbs the raw probabilities $\{p_\phi^c(t)\}_{c=1}^{C}$ with Gumbel-distributed noise, then takes a softmax with a temperature that aneals toward 0. At 0 temperature, Gumbel-Softmax exactly implements Eq. 4. When the temperature is positive, Gumbel-Softmax produces program architectures α whose constituent "one-hot" vectors actually contain multiple positive components. This causes the relaxed denotation operator to interpolate the behavior of nearby program architectures, yielding stable gradient estimation.

[1] To get intuition on why this assumption is reasonable, imagine that α is the contents of the input tape of a Universal Turing Machine with tape alphabet Σ; the UTM's output is α's denotation. If C is the maximum program length, then we need C one-hot vectors of dimension $\mid \Sigma \mid$ to encode α, because α would be represented by a length C string.

Expression: $(x + x) \times y$
Domain Specific Language:
$f_+(a, b) = a + b \qquad f_\times(a, b) = a \times b$
Straight Line Code:
$\ell_1 \leftarrow x$
$\ell_2 \leftarrow y$
$\ell_3 \leftarrow f_+(\ell_1, \ell_1)$
$\ell_4 \leftarrow f_\times(\ell_3, \ell_2)$
Architecture parametrization:

	function		left arg			right arg		
	f_+	$f_\times$	ℓ_1	ℓ_2	ℓ_3	ℓ_1	ℓ_2	ℓ_3
Line 3	(1,	0)	(1,	0)		(1,	0)	
Line 4	(0,	1)	(0,	0,	1)	(0,	1,	0)

Figure 2. Symbolic expressions are built from operators in a Domain Specific Language and represented as straightline code. Each line of code is parametrized by three one-hot binary vectors specifying a function from the Domain Specific Language, and left/right arguments from earlier lines. (The first lines of code simply load variables into scope.) The bottommost box shows the 6 one-hot vectors encoding the example expression (α in the paper).

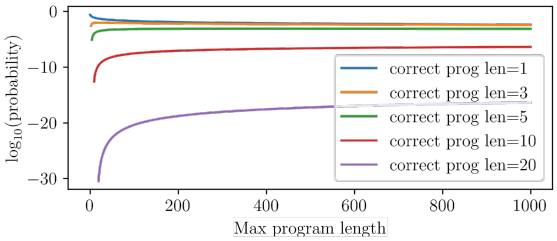


Figure 3. Simulation results showing probability of randomly initializing to a correct program, while varying max program length and target program length

(Over)parameterizing the program space. We now specify what program architectures look like, and how we parameterize them in terms of one-hot vectors. We model each program architecture as straightline code: A sequence of L lines of code, each of which introduces a new variable in scope by applying a function to variables introduced by preceding lines of code (Fig. 2). Each function comes from a Domain Specific Language, which contains components customized to the kinds of programs we expect to synthesize. Toggling which vector component of α is a 1 corresponds to toggling which function each line of code uses, and which preceding lines are passed as arguments to that function.

To compute the denotation of α, given its vectorized encoding, we use a simple dynamic program that memoizes the computation of the value computed by each line of code. This runs in time quadratic w.r.t. the total lines of code.

Although this parametrization works reasonably well, there are many alternatives. We also tried encoding a syntax tree instead of a list of lines of code, but this worked worse. In general, the classic program synthesis literature is filled with different techniques for 'sketching' a large set of possible programs, and then indexing that set with boolean decision variables (Solar Lezama, 2008; Jha et al., 2010).

We now **overparametrize** the problem by expanding the maximum possible lines of code far beyond what the system needs to solve its programming problems. This dramatically increases the dimensionality of α, and empirically we found that this significantly improved the convergence properties of gradient descent when optimizing Eq. 4. Without overparametrization, the system is prone to falling into poor local minima. We speculate that overparametrization helps for our problem for similar reasons as to why it helps for deep networks: (1) that it is harder to get trapped in higher dimensional spaces, because there is likely at least one direction which leads to lower loss, and (2) with more parameters there is a higher chance of a randomly initialized subnetwork falling within the basin of a good optimum, known as the lottery ticket hypothesis (Frankle & Carbin, 2018).

Speculatively, if lottery-ticket type behavior accounts for the success of overparametrization in our setting, then we might expect that increasing the maximum lines beyond the needed sizes actually *increases* the probability that randomly initialized weights encode the correct program. We built a simplified theoretical model of randomly initialized program architectures (Appendix Sec. D). Using this model we calculated the probability of a random network containing the correct program (Fig. 3). Across a range of different ground-truth program lengths, this probability saturates around a few tens of lines of code. In agreement with this analysis, we empirically found $L = 30$ max lines of code worked well on both of our domains.

Regularizing program length. Unsurprisingly, overparametrizing by increasing the max program size generates excessively long programs. Because length is a proxy for complexity, these programs might also tend to be more overfit, and also harder for humans to understand and interpret.

To combat the code explosion caused by overparametrization, we incorporate an additional term in our loss which penalizes the average program length. Calculating program length from our parametrization of α is straightforward to do in linear time using dynamic programming, and is also a smooth, differentiable function of α's components. Hence we can simply add the length-penalizing term to our loss.

In practice we train ROAP without regularization for the first half of its training process–to encourage exploration–and then turn on this regularizer halfway through to compress and optimize the programs (Fig. 4).

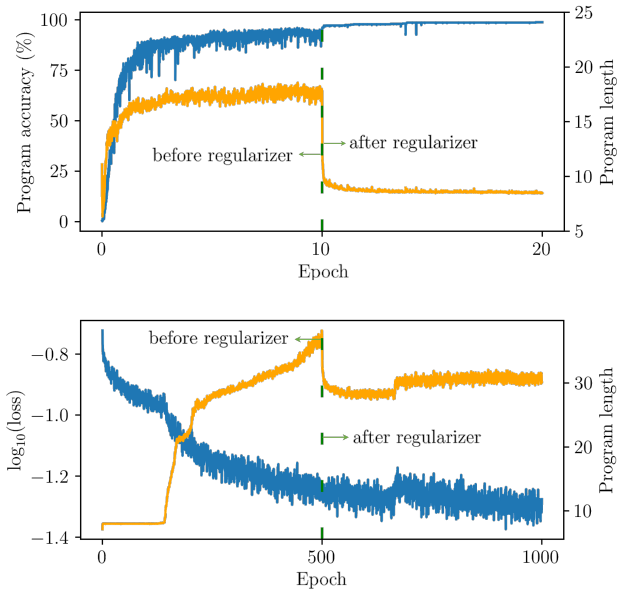
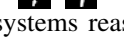


Figure 4. Regularizing program length halfway through training refactors the programs to be shorter (orange) without sacrificing accuracy (blue). Top: CIFAR-MATH. Bottom: 3D graphics.

Min-sampling. The policy acts as a search heuristic that stochastically proposes programs. Running the policy multiple times per task and taking the sampled program with the minimum loss allows trading more compute for lower loss, a trick we call 'Min-sampling'. This is conceptually related to importance reweighting of samples from neural recognition models (Burda et al., 2015).

4. Experiments

4.1. CIFAR-MATH

The classic warmup problem for neurosymbolic systems is to train an MNIST classifier by supervising only on the result of running an algorithm on that classifier's outputs. For example, DeepProbLog (Manhaeve et al., 2018) and Scallop (Huang et al., 2021) both train a digit classifier given examples of handwritten digits being added together: Given examples like  →11, together with the logic of addition, these systems reason backward through the addition operator to train a neural network MNIST classifier.

CIFAR-MATH makes this warmup domain harder along several dimensions. First, we introduce program synthesis by not telling the system what arithmetic expression is executing on the input images: this system must infer and synthesize the correct symbolic equation. Second, we consider more complex equations with several arithmetic operators. Last, we switch from MNIST to CIFAR-10, and do not tell the system that there are only 10 digits. Thus the system has a harder reasoning challenge, because it has to reason backward through more complex expressions; a new induction challenge, because the expressions are hidden; and a nontrivial perception challenge, because CIFAR-10 is more visually complex than MNIST.

Following Fig. 1, each task has a different *hidden equation* (Fig. 1 illustrates $x + yz$). The inputs to the equation are presented as CIFAR-10 images. Each of the ten CIFAR-10 categories (dog, boat, frog, ...) is mapped to a different digit from 0–9, but this mapping is never given to the system. Architectures α are built from a Domain Specific Language containing addition, multiplication, and subtraction. The shared continuous parameters θ are the weights of a CNN that maps a CIFAR-10 image to a scalar.

We are interested in a variety of research questions, and compute evaluation metrics to help us answer each of them:

- Can we synthesize the correct program? Because CIFAR-MATH comes with ground truth hidden programs, we check if the synthesized programs generate the same outputs on random inputs.
- Do we successfully learn a neural perception module, equivalently did the CNN learn a CIFAR-10 classifier as a side effect of the overall training procedure? To evaluate this we snap the CNN outputs to the nearest integer and report how often this yields the correct integer. For example, if frogs correspond to the number 3, then correctly classifying a frog means predicting a number in the range $[2.5, 3.5)$.
- To understand if the learned programs generalize out of sample, we designate one task to be trained only on small numbers (0–5), and then check if the synthesized neurosymbolic program extrapolates to larger numbers (6–9; multitasking makes it see these numbers on other tasks).

Tbl. 1 shows the metrics relating to the above questions for our system as well as ablations and baselines. We use a **REINFORCE** baseline, which uses the score function estimator instead of Gumbel-Softmax; **NEAR** (Shah et al., 2020), which uses A* to search the space of neurosymbolic programs, and does not perform multitasking or amortized inference; **dPads** (Cui & Zhu, 2021), which improves upon NEAR; **Terpret** (Gaunt et al., 2016), which directly optimizes the parameters of α, and does not perform multitasking or amortized inference ; and **HOUDINI** (Valkov et al., 2018), which solves tasks sequentially via enumeration while sharing neural network parameters across tasks.

Overall, we find that the full model can jointly learn to ground its visual input into discrete symbols (numbers 0–9), and then transform those discrete symbols using symbolic

Table 1. Experimental Results on CIFAR-MATH. min-sampling: 3 samples/gradient step

	Program-Acc	Symbolic-Acc	Test-Symbolic-Acc	Loss	Test-Loss	OOD-Loss
ROAP (ours)	91.8%	82.6%	48.7%	0.005	**0.11**	0.07
+ min-sampling	**99.8%**	**100%**	**69.4%**	2.4e-4	**0.11**	**0.05**
+ min-sampling; contiguous-image	**99.2%**	**100%**	**72.2%**	7.3e-4	**0.12**	**0.04**
w./o. program	0.0%	4.2%	4.7%	**1.3e-5**	**0.11**	0.52
w./o. amortized inference	28.4%	49.5%	27.9%	0.022	0.16	0.15
w./o. gumbel-softmax	21.0%	1.8%	0.9%	0.007	0.14	2.07
w./ Syntax-Tree parametrization	69.0%	43.0%	28.1%	0.013	0.14	0.22
w./ max lines=10	11.6%	9.9%	6.9%	0.027	0.14	2.12
REINFORCE	0.2%	0.0%	0.0%	0.59	0.65	0.64
Terpret	0.9%	4.0%	3.4%	6.4e19	5.9e19	N/A
Terpret + multitasking	7.4%	11.1%	8.4%	7.6	12.8	0.74
NEAR	0.8%	8.3%	6.9%	0.059	0.18	0.93
dPads	2.0%	9.6%	9.2%	0.36	0.36	0.96
HOUDINI	17.4%	13.1%	9.4%	0.015	0.09	0.60

equations that the system itself infers. None of the baseline neurosymbolic synthesis methods meet that criteria.[2] We also find that a symbolic program aids out-of-sample generalization, as can be seen by comparing with the 'w/o program' baseline, which replaces the program architecture with a small neural network. This suggests ROAP has learned an appropriate division of labor between its CNN and its symbolic programs, with the CNN handling perception (but not reasoning) while the symbolic programs handle reasoning, thus enabling it to extrapolate out-of-distribution.

We additionally verified that ROAP does not need the image to be split into three separate images showing each digit. When the input is presented as a single contiguous image, our model's performance is essentially unchanged (Tbl. 1, 'contiguous-image').

Why does the model learn the 'right' latent symbols? A single CIFAR-MATH problem is ill-posed: it is not clear what latent symbols the neural network should output, because they are reprocessed by a (latent) program. Our experiments establish however that the system readily converges on the 'right' symbol grounding by mapping each CIFAR-10 image category to its corresponding digit. Intuitively, this happens because multitasking introduces extra constraints on the function learned by the neural component, which has to serve a variety of downstream symbolic computations.

If this story is true, then the ability of the system to converge on a good symbol grounding hinges on having a sufficiently constrained optimization problem. Extra tasks impose extra constraints, but so does having more input-output examples for each task. We therefore study whether either of those constraints suffice for recovering the correct symbol grounding. Fig. 5 shows success in recovering the correct symbolic basis as a function of the constraints imposed on the optimization problem, both by multitasking and input-outputs per task. We see a phase-transition like structure where, once the total number of constraints passes a tipping point, the system 'snaps' into the expected symbolic basis. This shows the importance of constraints, and also that there is a tradeoff between the number of tasks, and the number of examples per task.

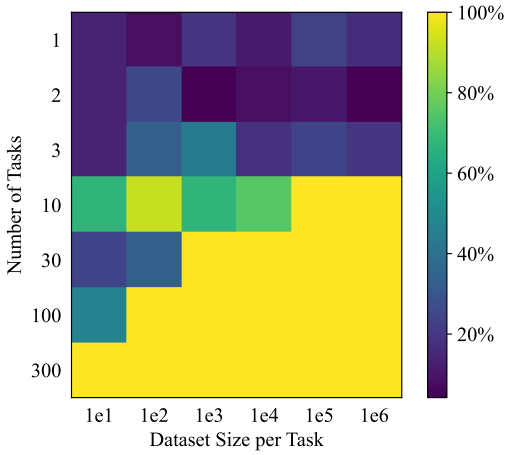


Figure 5. Effects of constraints on symbol grounding. Both having more tasks, and having more input-outputs per task, introduced added constraints. Heatmap shows max Symbolic-Acc over 5 runs, and should be approximately interpreted as whether or not it has a good chance of converging correctly in 5 runs.

[2]We also tried Scallop (Huang et al., 2021), a leading neural logic programming system, but its differentiable-top-k-proofs inference method did not terminate on CIFAR-MATH problems. We suspect this is because it has to build a massive proof tree when the arithmetic equation is unknown.

Table 2. 2D results. (NEAR's pathological behavior on these problems is to loop forever because it can fit a shape arbitrarily well with increasingly long partially completed programs, thus it never terminates with a completed program. HOUDINI's enumeration is inapplicable due to continuous parameters in affine transforms)

Method			Chamfer Distance	
Name	Train-mode	Test-beam-size	No-refinement	Test-time-refinement
CSG-Net	Supervised+RL	10	1.14	0.41
CSG-NETSTACK	Supervised+RL	10	1.02	0.34
PLAD	LEST+ST+WS	10	0.811	-
UCSGNet	Unsupervised	1	0.32	-
REINFORCE	RL	1	inf	-
NEAR	Unsupervised	1000	N/A	Pathological behavior
HOUDINI	Unsupervised	-	N/A	N/A
Terpret	Unsupervised	1	N/A	4.76±2.22
ROAP (ours)	Unsupervised	1	**0.21**	-

4.2. Graphics Program Synthesis

We use ROAP to synthesize neurosymbolic graphics programs. We consider the problem of *reconstruction*, which means inferring the shape of an object given a partial/occluded observation. The graphics programs start with basic parts, like boxes and balls, which are transformed and combined to generate 3D geometry. What makes these graphics programs neurosymbolic is that, instead of hard-coding these basic parts, we allow the system to learn its own part library. Each learned part is a simple shape that can be viewed as the output of a (tiny and unusual) neural network, whose parameters comprise θ (Appendix Sec. C)

Our Domain Specific Language for graphics programs includes the ability to intersect and union shapes; reflect shapes over principal axes; a `for` loop that repeatedly translates its loop body; and affine transformations upon basic parts. Each graphics program is a function from a point in space ($\mathbb{R}^3$) to a boolean indicating whether that point is inside or outside of the object.

We first test on reconstructing 2D silhouettes of furniture (Fig. 6), which a series of recent graphics program synthesis works evaluate on (Jones et al., 2022; Sharma et al., 2022; Kania et al., 2020; Sharma et al., 2018); see Appendix B.2. We assess reconstruction accuracy via Chamfer distance between the ground truth shape and the output of the synthesized program. Tbl. 2 shows that ROAP achieves higher reconstruction accuracy compared to these comparable recent works.

Next we evaluate on 3D models. Because ROAP does not supervise on ground-truth programs, we can apply it to datasets not designed for program synthesizers, and so we choose the canonical ShapeNet dataset (Chang et al., 2015); see Appendix C.2. Our goal is to study the qualitative behavior of our system and contrast with other general-purpose program synthesizers, *not* to set a new state-of-the-art for ShapeNet. ShapeNet has received over 7 years

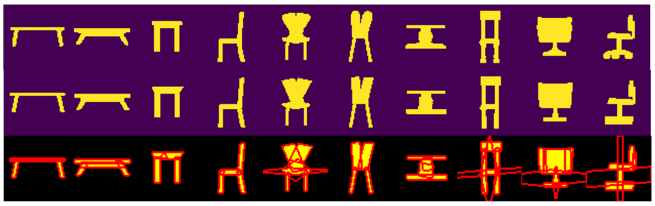


Figure 6. Qualitative results of 2D furniture silhouettes. Top, input silhouette. Middle, reconstruction. Bottom, parts used.

Table 3. Experimental Results on 3D

	Full	Crop-Plane
ROAP (ours)	1.7	1.8
w./o. program	1.2	2.0
w./o. amortized inference	N/A	N/A
w./o. gumbel-softmax	9.5	8.4
w./ Syntax-Tree	2.0	7.1
w./ depth=10	8.7	2.7
w./ depth=3	2.7	13.1

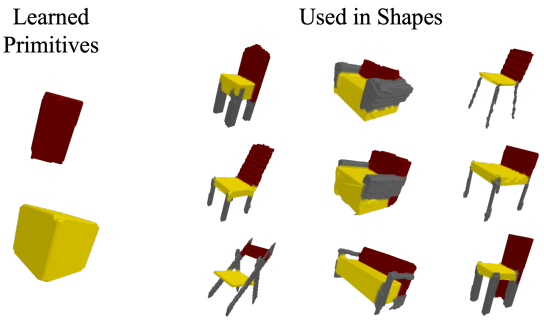


Figure 7. Neural part learning for 3D

of attention from the deep learning and computer vision communities, who have built sophisticated yet specialized 3D reconstruction networks and training regimes (Genova et al. (2020) is representative in its sophistication).

Each reconstruction task is specified by a partial observation of a voxel field, and the synthesized program describes how to generate the underlying complete 3D shape. We consider corrupting the input voxel field by cropping out a random half-plane (Fig. 8). Fig. 1 diagrams how ROAP represents shapes as programs that algebraically combine basic parts. Fig. 7 illustrates example learned parts: Rather than preprogram boxes, cylinders, etc., the system learns from the data which primitives are most suitable. It learned, for example, a boxy cuboid with rounded corners for modeling chair/sofa seats (yellow), and an elongated cuboid with a subtly curved top for modeling backs and headrests (red). This data-driven discovery of basic symbolic abstractions was done without supervising on programs or part decompositions.

Last, we again quantify reconstruction quality via Chamfer distance, and compare against ablations of our system (Tbl. 3). The most important ablation of our system is 'w/o program'. This replaces the program with a neural network, essentially modeling an occupancy network (Mescheder et al., 2019), which is a foundational deep learning architecture for 3D reconstruction. Given a complete observation of the shape, a pure neural network is superior at reconstruction ('Full' in Tbl. 3). But given a partial observation, the neurosymbolic program comes out ahead ('Crop-Plane' in Tbl. 3). This is because the symbolic program structure has an inductive bias primed to recognize symmetries and repeated parts. Hence this high-level symbolic prior helps impute missing observations.

5. Related Work

Neurosymbolic programming is a growing area that seeks to engineer learning and inference methods for hybrid program/neural architectures (Chaudhuri et al., 2021), and our work is a special case of this broad framework. Specifically, we tackle inductive program synthesis (Gulwani et al., 2017)–synthesizing programs from input-output examples–but where the inputs are continuous and must be preprocessed by neural networks into symbolic form. Prior works in this setting assume a hand-engineered inventory of basic symbols (Ellis et al., 2018), while others backpropagate through differentiable programs to jointly train network weights and program structure (Gaunt et al., 2017). Mul-

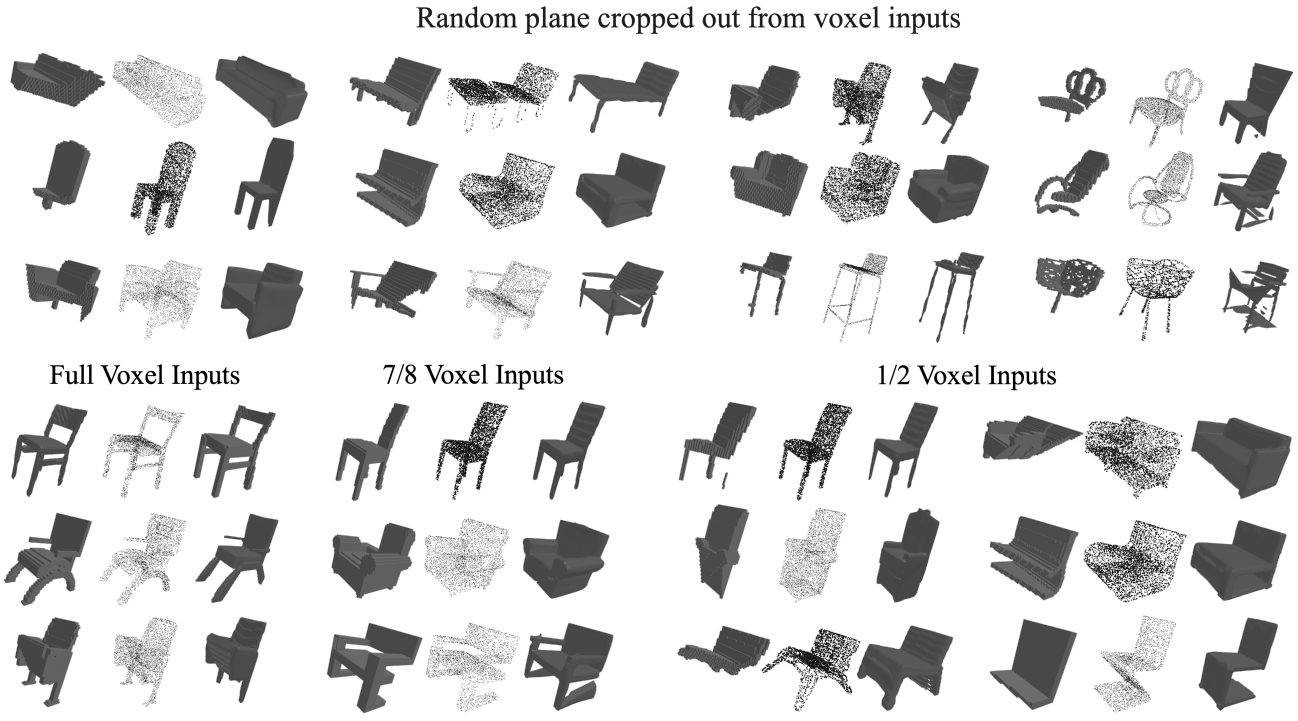


Figure 8. Results on 3D where the system inputs a voxel field with a random subset cropped out (left models), from which it synthesizes a program (right models) that approximates the ground-truth shape (middle point cloud). Our model can be trained to complete partial geometry with a variety of different subsets taken out. Cropping out more of the input voxels makes the problem harder. Tbl. 3 reports quantitative results contrasting the easiest regime (full voxel inputs) vs the hardest regime (an entire random half plane cropped).

titasking is a known strategy for this setting (Valkov et al., 2018). Training a neural network to help guide search for discrete programs (amortized inference) is standard (Shi et al., 2021; Chen et al., 2018), and we extend that idea to continuous relaxations of program spaces.

The difficulties of gradient descent over relaxed program spaces is well known, to the extent that it has been dubbed the so-called 'terpret problem' (Gaunt et al., 2016). Unfortunately, such an approach is the most straightforward way of training neurosymbolic programs. From a technical perspective, our work hopes to make progress on the 'terpret problem', thereby unlocking scalable and reliable training of this class of neurosymbolic programs.

Some of our core tricks have debuted in prior neurosymbolic program synthesizers: Terpret noticed overparametrization helps (Gaunt et al., 2016), Memoized Wake-Sleep deployed amortized inference (Hewitt & Tenenbaum, 2019), and HOUDINI shares neural network parameters across tasks (Valkov et al., 2018). Our technical contribution is providing the mathematical and algorithmic framework which allows these tricks, and more, to be combined into the same end-to-end learnable system. For example, we showed that the reparametrization trick (Jang et al., 2016) made amortization compatible with relaxation and gradient-guided search.

More fundamentally, our efforts connect to the body of work on the 'symbol grounding problem' (Harnad, 1990): How does a system learn to 'ground' abstract symbols (e.g., numbers, parts) in terms of their high-dimensional perceptual counterparts (e.g., images of digits)? This problem is especially difficult absent strong supervision on the meaning of each abstract symbol (Chang et al., 2020), and ROAP considers a distantly supervised setting. Prior works consider a variety of orthogonal techniques to address symbol grounding (Topan et al., 2021), including scaffolding with natural language (Andreas et al., 2016; Mao et al., 2021).

6. Conclusion

Our goal is to make progress on basic neurosymbolic problems: starting from perception, and absent symbol-level supervision, how can we discover basic symbolic abstractions together with the symbolic programs which manipulate them? Although our experiments confirm that ROAP might be on the right track for solving these problems, our method has important limitations. ROAP cannot operate without a reasonably-sized training corpus of programming tasks, although the fact that it does not need to supervise on source code helps address this limitation. Fundamentally, ROAP assumes end-to-end gradient descent is the right approach, which means that program execution must be relaxed and differentiated. It is not clear that differentiable program induction can handle sophisticated programming constructs, such as data structures and recursion (Feser et al., 2017), at least in its current form. Thus we especially hope ROAP helps spur more fundamental progress on differentiable program relaxation techniques.

Acknowledgements. We gratefully acknowledge the support of a Research Scholar gift from Google.

References

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 39–48, 2016.

Burda, Y., Grosse, R., and Salakhutdinov, R. Importance weighted autoencoders. *arXiv preprint arXiv:1509.00519*, 2015.

Chang, A. X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., Xiao, J., Yi, L., and Yu, F. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.

Chang, O., Flokas, L., Lipson, H., and Spranger, M. Assessing satnet's ability to solve the symbol grounding problem. *Advances in Neural Information Processing Systems*, 33:1428–1439, 2020.

Chaudhuri, S. and Solar-Lezama, A. Smooth interpretation. *ACM Sigplan Notices*, 45(6):279–291, 2010.

Chaudhuri, S., Ellis, K., Polozov, O., Singh, R., Solar-Lezama, A., and Yue, Y. Neurosymbolic programming. *Foundations and Trends® in Programming Languages*, 7(3):158–243, 2021. ISSN 2325-1107. doi: 10.1561/2500000049. URL `http://dx.doi.org/10.1561/2500000049`.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models

trained on code. *CoRR*, abs/2107.03374, 2021a. URL https://arxiv.org/abs/2107.03374.

Chen, Q., Lamoreaux, A., Wang, X., Durrett, G., Bastani, O., and Dillig, I. Web question answering with neurosymbolic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 328–343, New York, NY, USA, 2021b. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454047. URL https://doi.org/10.1145/3453483.3454047.

Chen, X., Liu, C., and Song, D. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

Chen, Z., Tagliasacchi, A., and Zhang, H. Bsp-net: Generating compact meshes via binary space partitioning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 45–54, 2020.

Cui, G. and Zhu, H. Differentiable synthesis of program architectures. *Advances in Neural Information Processing Systems*, 34:11123–11135, 2021.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. *ICML*, 2017.

Ellis, K., Ritchie, D., Solar-Lezama, A., and Tenenbaum, J. B. Learning to infer graphics programs from hand-drawn images. *NIPS*, 2018.

Evans, R. and Grefenstette, E. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018.

Feser, J. K., Brockschmidt, M., Gaunt, A. L., and Tarlow, D. Neural functional programming. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=Byp_ccVte.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Gaunt, A. L., Brockschmidt, M., Singh, R., Kushman, N., Kohli, P., Taylor, J., and Tarlow, D. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

Gaunt, A. L., Brockschmidt, M., Kushman, N., and Tarlow, D. Differentiable programs with neural libraries. In *International Conference on Machine Learning*, pp. 1213–1222. PMLR, 2017.

Genova, K., Cole, F., Sud, A., Sarna, A., and Funkhouser, T. Local deep implicit functions for 3d shape. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4857–4866, 2020.

Gershman, S. and Goodman, N. Amortized inference in probabilistic reasoning. In *Proceedings of the annual meeting of the cognitive science society*, volume 36, 2014.

Gulwani, S., Polozov, O., Singh, R., et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4 (1-2):1–119, 2017.

Harnad, S. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1-3):335–346, 1990.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hewitt, L. and Tenenbaum, J. Learning structured generative models with memoised wake-sleep. *under review*, 2019.

Huang, J., Li, Z., Chen, B., Samel, K., Naik, M., Song, L., and Si, X. Scallop: From probabilistic deductive databases to scalable differentiable reasoning. In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 25134–25145. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/file/d367eef13f90793bd8121e2f675f0dc2-Paper.pdf.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. Oracle-guided component-based program synthesis. In *ICSE*, volume 1, pp. 215–224. IEEE, 2010.

Jones, R. K., Walke, H., and Ritchie, D. Plad: Learning to infer shape programs with pseudo-labels and approximate distributions. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

Kania, K., Zieba, M., and Kajdanowicz, T. Ucsg-net-unsupervised discovering of constructive solid geometry tree. *Advances in Neural Information Processing Systems*, 33:8776–8786, 2020.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., and De Raedt, L. Deepproblog: Neural probabilistic

logic programming. In *Advances in Neural Information Processing Systems*, pp. 3749–3759, 2018.

Mao, J., Shi, F., Wu, J., Levy, R., and Tenenbaum, J. Grammar-based grounded lexicon learning. *Advances in Neural Information Processing Systems*, 34:7865–7878, 2021.

Mescheder, L., Oechsle, M., Niemeyer, M., Nowozin, S., and Geiger, A. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Sahoo, S., Lampert, C., and Martius, G. Learning equations for extrapolation and control. In *International Conference on Machine Learning*, pp. 4442–4450. PMLR, 2018.

Shah, A., Zhan, E., Sun, J., Verma, A., Yue, Y., and Chaudhuri, S. Learning differentiable programs with admissible neural heuristics. *Advances in neural information processing systems*, 33:4940–4952, 2020.

Sharma, G., Goyal, R., Liu, D., Kalogerakis, E., and Maji, S. Csgnet: Neural shape parser for constructive solid geometry. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5515–5523, 2018.

Sharma, G., Goyal, R., Liu, D., Kalogerakis, E., and Maji, S. Neural shape parsers for constructive solid geometry. *IEEE transactions on pattern analysis and machine intelligence*, 44(5):2628—2640, May 2022. ISSN 0162-8828. doi: 10.1109/tpami.2020.3044749. URL https://doi.org/10.1109/TPAMI.2020.3044749.

Shi, K., Dai, H., Ellis, K., and Sutton, C. Crossbeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations*, 2021.

Si, X., Raghothaman, M., Heo, K., and Naik, M. Synthesizing datalog programs using numerical relaxation. *IJCAI*, 2019.

Solar Lezama, A. *Program Synthesis By Sketching*. PhD thesis, 2008.

Topan, S., Rolnick, D., and Si, X. Techniques for symbol grounding with satnet. *Advances in Neural Information Processing Systems*, 34, 2021.

Valkov, L., Chaudhari, D., Srivastava, A., Sutton, C., and Chaudhuri, S. Houdini: Lifelong learning as program synthesis. In *Advances in Neural Information Processing Systems*, pp. 8687–8698, 2018.

A. CIFAR-MATH Experimental Details

A.1. Experimental Setup

Neural network. We use an 18-layer ResNet backbone (He et al., 2016) as the image encoder with an MLP decoder, whose parameters collectively comprise θ.

Program denotation. Fig. 9 specifies how we execute program architectures α in this domain using a simple dynamic program.

Dataset. We generate 500 arithmetic tasks with 3 input variables, containing up to 3 operators. For each arithmetic task we have 1e6 I/O pairs for each task for training and 1000 I/O pairs for each task for testing.

Training. We train models using the Adam (Kingma & Ba, 2014) optimizer with a learning rate equal to 3e-4 and ϵ =1e-5 for 20 epochs. The program length regularizer is not applied until halfway through training with a coefficient of λ =1e-4, which is multiplied into the program length before it is added to the rest of the loss. The temperature for gumbel softmax is set to 1 in the beginning and changed to 3 from epoch 15 to minimize the error gap from the continuous approximations of programs near the end of training.

$$[\![\alpha]\!]_\theta(x) = \text{Exec}_\theta(\alpha, x, L+V) \quad \textit{execute program and extract output on line } L+V$$

$$\text{Exec}_\theta(\alpha, x, l) = \text{CNN}_\theta(x_l), \text{whenever } l \leq V \quad \textit{load variables as first lines of code. We have V variables}$$

$$\text{Exec}_\theta(\alpha, x, l) = \sum_o \alpha^O_{lo} \times F_o\left(x, \sum_{1\leq a<l} \alpha^L_{la} \times \text{Exec}_\theta(\alpha, x, a), \sum_{1\leq b<l} \alpha^R_{lb} \times \text{Exec}_\theta(\alpha, x, b)\right), \text{whenever } l > V$$

where α is a tuple of $(\alpha^O, \alpha^L, \alpha^R)$

$F_1(x, A, B) = A + B$ *add*

$F_2(x, A, B) = A - B$ *subtraction*

$F_3(x, A, B) = A \times B$ *multiplication*

$F_4(x, A, B) = A$ *no-op/skip connection*

Figure 9. **Differentiable execution model** for a program sketch containing L lines of code. α parametrizes the program via a triple of 2-dimensional arrays $(\alpha^O, \alpha^L, \alpha^R)$ containing values from 0-1. If $\alpha^O_{lo} = 1$, then line l of the program computes its value by executing operator o. If $\alpha^L_{la} = 1$, then line l of the program gets its left argument for the operator from line a. If $\alpha^R_{lb} = 1$, then line l of the program gets its rights argument for the operator from line b. The first V lines of the program evaluate to input variables, and we assume that there are V such variables and L lines of code that follow.

B. 2D Reconstruction Experimental Details

B.1. Methods

B.1.1. CSG, Flip-Union, and For-Loop Operations

We now specify the denotation of graphics programs. Recall that every graphics program is a function that takes a point in space ($\mathbb{R}^3$) to 0/1 depending on if that point is outside or inside the object. In the relaxed semantics, we think of the denotation as producing a number in the range $[0, 1]$. We refer to such numbers in $[0, 1]$ as 'occupancy values'.

In general, the denotations of the graphics operations follows straightforwardly from their mathematical definitions. For example, the union operator is represented as the maximum of its argument's denotations, i.e., $o = \max(o_{\text{left}}, o_{\text{right}})$. Specifically, the occupancy function of any shape that is a union of two parts is defined as $[\![\bigcup(z_{\text{left}}, z_{\text{right}})]\!](\vec{p}) = \max([\![z_{\text{left}}]\!](\vec{p}), [\![z_{\text{right}}]\!](\vec{p}))$. The intersection operator is represented as the minimum of the occupancy values, i.e., $o = \min(o_{\text{left}}, o_{\text{right}})$, and the difference operator is represented as $o = \max(o_{\text{left}} - o_{\text{right}}, 0)$.

The flip-union and for-loop operations are implemented as unions of multiple sub-components. The occupancy values of the sub-components are determined by applying transformations to the coordinates that align with their semantics. Specifically, the flip-union operation is defined as:

$$[\![\text{flip-union}_{\theta_f}(z)]\!](\vec{p}) \quad = \quad \max([\![z]\!](\vec{p}), [\![z]\!](\text{flip}_{\theta_f}(\vec{p}))),$$

where $\theta_f \in \mathbb{R}^3$ defines a line by $\theta_f^T[\vec{p};\ 1] = 0$ in the 2D case. Flipping a point against this line, i.e., $\text{flip}_{\theta_f}(\vec{p})$ can be implemented using a simple affine transformation. Let $\theta_f = [a, b, c]$, then

$$\text{flip}(\vec{p}) = \frac{1}{a^2+b^2}\begin{bmatrix} b^2-a^2 & -2ab & 2ac \\ -2ab & a^2-b^2 & 2bc \end{bmatrix}\vec{p}.$$

The for-loop operation is defined as

$$[\![\text{for-loop}_{\theta_d, C}(z)]\!](\vec{p}) \quad = \quad \max(\{[\![z]\!](\vec{p} - c \times \theta_d) : c \in \{0, 1, \cdots, C-1\}\}),$$

which repeats the part $[\![z]\!]$ for C times by moving it in $\theta_d \in \mathbb{R}^2$ direction for c times.

B.1.2. Program Sketch

The generation program is a union of three components: a simple component, a symmetry component, and a repeated component. The simple component, denoted as $[\![z_{\text{sim}}]\!]$, comprises of various CSG operations. The symmetry component, $[\![\text{flip-union}_{\theta_f}(z_{\text{sym}})]\!]$, is implemented using the flip-union operator. The repeated component, $[\![\text{for-loop}_{\theta_d, C}(z_{\text{repeat}})]\!]$, is implemented using the for-loop operator. All sub-components $[\![z_{\text{sim}}]\!]$, $[\![z_{\text{sym}}]\!]$, and $[\![z_{\text{repeat}}]\!]$, involve multiple CSG operations on simple primitive shapes such as squares and circles, using the straight-line-coding formulation. In addition, we incorporate gate parameters $\theta_{\alpha_{\text{sym}}}$ and $\theta_{\alpha_{\text{repeat}}}$ to control which components are included as

$$\begin{aligned}
[\![\text{shape-program}]\!](\vec{p}) &= \max([\![z_{\text{sim}}]\!](\vec{p}), \alpha_{\text{sym}} \times [\![\text{flip-union}_{\theta_f}(z_{\text{sym}})]\!](\vec{p}), \alpha_{\text{repeat}} \times [\![\text{for-loop}_{\theta_d, C}(z_{\text{repeat}})]\!]) \\
\alpha_{\text{sym}} &= \mathbf{1}(\theta_{\alpha_{\text{sym}}} >= 0) \\
\alpha_{\text{repeat}} &= \mathbf{1}(\theta_{\alpha_{\text{repeat}}} >= 0).
\end{aligned}$$

B.1.3. Balanced Training Loss

To prevent models from becoming stuck in sub-optimal local optimums that fit only a portion of the training data, we employ a balanced training loss that adjusts the weights of samples based on the model's performance on them. Specifically, the loss is designed to as

$$\mathcal{L}(\theta, \phi) \quad = \quad \mathbb{E}_{t\sim\mathcal{T}}\left[\text{weight}(t;\ \theta, \phi) \sum_{(x,y)\in\mathcal{D}_t} \text{Loss}(x, y;\ \theta, \phi)\right].$$

We use the chamfer distance to measure the performances of models, $\text{cd}(t;\ \theta, \phi)$, and adjusts the weights of samples accordingly: $\text{weight}(t;\ \theta, \phi) = 1 - \max\left(1 - \frac{\text{cd}(t;\ \theta,\phi)}{\text{threshold}}, 0\right)$. In practice, we set the threshold to be 0.95 based on our preliminary experimental results.

B.2. Experimental Setup

We utilize the same CNN encoder and MLP decoder as UCSGNet (Kania et al., 2020). The program sketch includes one simple component, two symmetry components, and one repeated component. Each of their sub-components has 20 lines of codes in addition to the 32 transformed primitive shapes, including 16 circles and 16 squares. The maximum count of the for-loop operator is 3. We use the same dataset as UCSGNet (Kania et al., 2020) which consists of 8000 CAD shapes in three categories, chair, desk, and lamps (Sharma et al., 2018).

B.3. More Experimental Results

More reconstruction results are visualized in Figure 10.

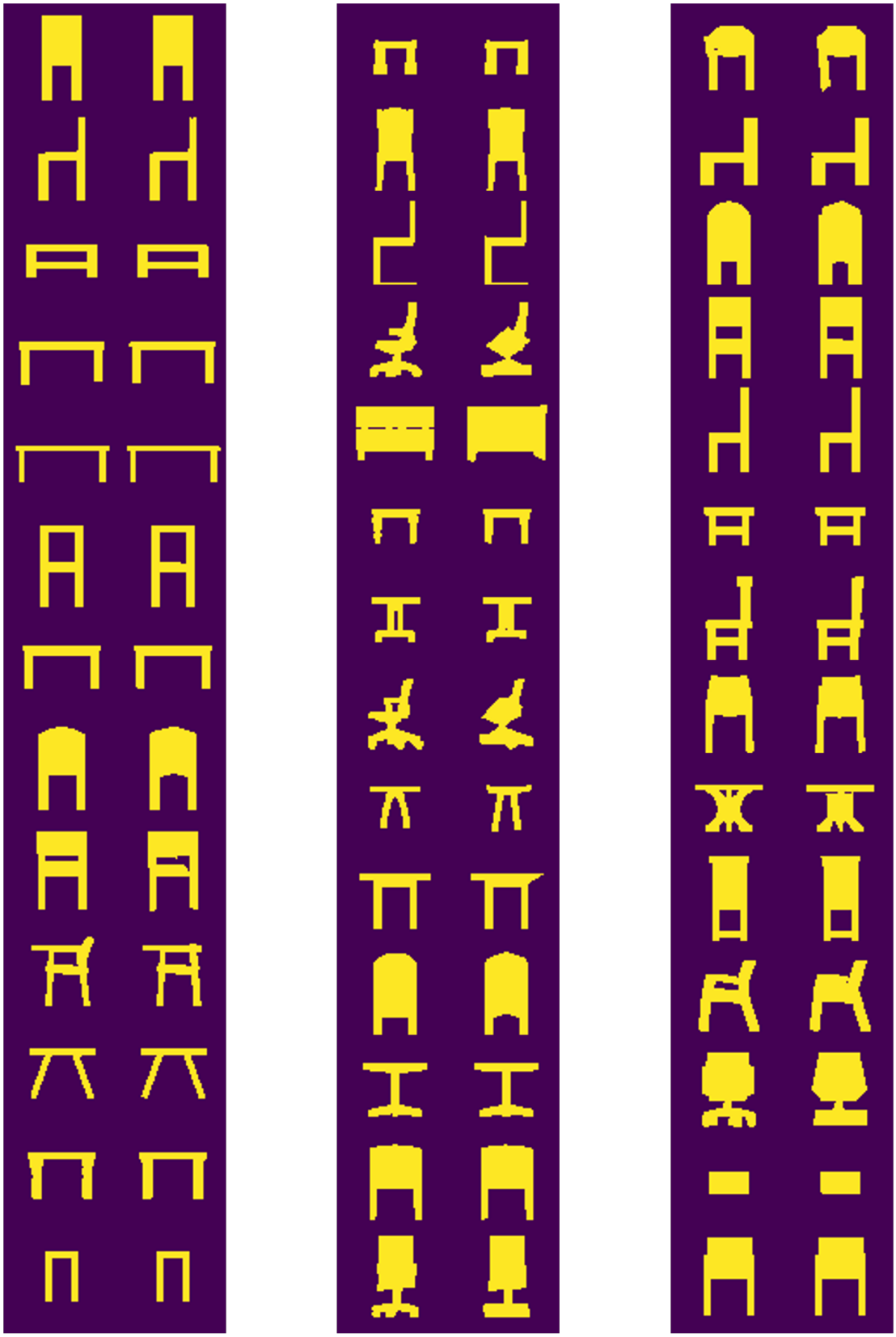


Figure 10. 2D Results

C. 3D Reconstruction

C.1. Method

C.1.1. Quadratic Primitive Shapes

Our program's expressive capacity is enhanced by the integration of curved primitive shapes, defined using a combination of two quadratic functions. The first function dictates the surface geometry, while the second performs a warp transformation. The signed distance function that defines the surface geometry is presented as follows:

$$\mathcal{D}_{\theta_g}(\vec{p}) = \max(\theta_g^T[p_x^2, p_y^2, p_z^2, |p_x|, |p_y|, |p_z|] - 1,\ \ 2|p_x| - 1,\ \ 2|p_y| - 1,\ \ 2|p_z| - 1).$$

This formulation ensures that the primitive shapes do not exceed the dimensions of a $1 \times 1 \times 1$ box by utilizing the last three linear surfaces. Additionally, the use of absolute value functions guarantees symmetry across the x, y, and z planes. This formulation not only allows for the representation of basic shapes such as boxes and spheres, but also enables the representation of more complex, curved primitive shapes with greater representation power.

Similar to affine transformations, quadratic warp transformations f_{θ_w} can be represented using a coordinate mapping function, as demonstrated below:

$$\begin{aligned} f_{\theta_w}(\vec{p})_x &= \frac{p_x - t_x}{s_x} \\ f_{\theta_w}(\vec{p})_y &= p_y \\ f_{\theta_w}(\vec{p})_z &= p_z, \end{aligned}$$

where

$$\begin{aligned} \theta_w &= [\theta_s;\ \theta_t;\ \theta_\alpha] \\ \alpha_x &= \mathbf{1}(\theta_\alpha >= 0) \\ s_x &= \alpha_x \cdot (\theta_s^T[p_y, p_z, p_y^2, p_z^2, p_y p_z]) + 1 \\ t_x &= \alpha_x \cdot (\theta_t^T[p_y, p_z, p_y^2, p_z^2, p_y p_z]). \end{aligned}$$

The warp transformation allows for the representation of irregular shapes, such as the mattock shape depicted in Figure 1, by applying quadratic transformations to the coordinate x based on the coordinates y and z. The parameter α_x serves as a gate function that controls the degree of transformation. Note that, due to the symmetry properties of quadratic surfaces, the transformation of x is equivalent to transforming y and z.

C.1.2. Shape Library

To enhance the efficiency of primitive learning and grounding, we incorporate a shape library that is shared across tasks. The library comprises 128 warp transformations and 128 quadratic surface formulations, resulting in a total of 16,384 primitive shapes. The shape library retrieval mechanism is designed similarly to vector quantization. In particular, the shape library include parameters $\Theta_w \in \mathbb{R}^{128 \times |\theta_w|}$ for warp transformations and $\Theta_g \in \mathbb{R}^{128 \times |\theta_g|}$ for quadratic surfaces. For each query $\vec{q} \in \mathbb{R}^{|\theta_g| + |\theta_w|}$, it will return a primitive shape with the warp transformation parameter $\Theta_w\,[\arg\max \Theta_w \vec{q}\,[: |\theta_w|]]$ and the quadratic surface parameter $\Theta_g\,[\arg\max \Theta_g \vec{q}\,[-|\theta_g| :]]$. To enable training via gradient descent, we implement a probabilistic relaxation of the retrieval mechanism:

$$\begin{aligned} \alpha_w &= \text{(gumbel-)softmax}(\Theta_w \vec{q}[: |\theta_w|]) \\ \theta_w &= \alpha_w^T \Theta_w \\ \alpha_g &= \text{(gumbel-)softmax}(\Theta_p \vec{q}[-|\theta_g| :]) \\ \theta_g &= \alpha_g^T \Theta_g. \end{aligned}$$

The implicit function of the softly retrieved primitive shape is then $\mathcal{D}_{\theta_g}(f_{\theta_w}^{-1}(\vec{p}))$.

C.1.3. CSG AND FLIP-UNION OPERATIONS

The CSG operations are defined in the same way as in the 2D case as outlined in Appendix B.1.1. The flip-union operation is an extension of its 2D counterpart, with the added dimension of flipping points against a plane defined by $\theta_f[\vec{p};\ 1] = 0$. In practice, we set $\theta_f[-1] = 0$ to ensure that the plane passes through the origin.

C.1.4. PROGRAM SKETCH

The shape generation program is a symmetry of a sub-component that involves multiple CSG operations on learned primitives retrieved from a shared shape library. The CSG operations are formulated as straight-line coding.

C.2. Experimental Setup

We utilize the same 3D-CNN encoder and MLP decoder as UCSGNet (Kania et al., 2020). The program sketch is a symmetry of a sub-component that has 20 lines of code in addition to the 128 learned primitive shapes, retrieved from the shared shape library. We use the preprocessed dataset ShapeNet provided by (Chen et al., 2020). It includes 64^3 volumes of voxelized shapes and samples 16384 points as a ground truth with a higher probability of sampling near the surface for training. In the 7/8 voxelized input setting, we randomly crop one-octant of the voxels by setting their values to zero. In the 1/2 voxelized input setting, we randomly sample planes as $a, b, c \sim \text{uniform}(0, 1)$ and setting half of the voxels to zero, i.e., those voxels with coordinates $\vec{p}$ such that $ap_x + bp_y + cp_z > 0$.

D. Theoretical Setting

We demonstrate the advantages of using over-parameterization in a simplified setting. Specifically, we consider a program synthesis algorithm that utilizes random search, and we assume that all programs are distinct and that operators take only one argument. Our goal is to find a program in the form of $p = o_L(o_{L-1}(\cdots(o_1(x))))$, where $L \geq 1$ is the length of the program, $o_l \in \mathcal{O}$ denotes operators, and x is the input. There is no other program $p' \in \mathcal{P}$ such that $\forall\, x, p(x) == p'(x)$.

We represent the program sketch as a straight-line code with length L', as shown in Figure 2. Each line randomly selects one operator from the operator space $\mathcal{O}'$ and its argument from previous lines. We allow the program sketch to use the identity operator, resulting in $\mathcal{O}' = \mathcal{O} \bigcup \{\text{identity}\}$.

Due to the formulation of straight-line codes and the usage of the identity operator, there are multiple possible assignments of program parameters for the correct program. We calculate the exact probability of finding the correct program by randomly initializing the program assignment of the program sketch using dynamic programming. We set $|\mathcal{O}| = 3$ as the other experiments and show the probabilities of randomly initializing to a correct program for different correct program lengths L and program sketch lengths L' in Figure 3.